The Open Linux-On-Chip System

Author
Mario Boikov

Supervisor
Daniel Mattsson
Realway AB

Examiner
Per Andersson
LTH

Master's thesis
Lund Institute of Technology
Department of Computer Science
September 30, 2008

Abstract

Implementing SoCs based on FPGAs can be favourable since FPGAs are very flexible and versatile. Using Linux in embedded systems is common nowadays and the major FPGA vendors provides solutions to deploy Linux on FPGA-based SoCs, but almost all solutions are based on closed hardware. It can be preferable to not only use open software when creating systems but also use open hardware to create a complete open system, in other words an "Open Linux-On-Chip system". One advantage of using open hardware (and software) is the freedom from vendor lock-in, your solution will not be dependent on support and services from one company. Visibility is another advantage, you can verify correctness and get a detailed understanding of the architecture.

In this thesis I evaluate and investigate three options to run Linux on open hardware. The options are based on the soft-core processors LatticeMico32 from Lattice, LEON3 from Gaisler Research and OpenRISC 1200 from OpenCores. I found that the solution from Gaisler has the widest range of bundled peripherals, is most portable and includes technical features the other solutions misses. Lattice's solution is very user-friendly and easy to start with. The down-side is that it only works on chips from Lattice and only supports uClinux. OpenCores' solution requires a lot of manual work by the user and lacks a decent build system.

Finally I present a proof-of-concept project where the solution from Gaisler is extended with a hardware device and a Linux device driver. The custom made hardware and software are integrated into the respective build system provided by Gaisler and SnapGear Linux. This is done to investigate how extendable the solution is. It turns out that it is very easy to integrate both the custom made hardware and software into the respective build system.

Preface

During my studies at the university I became interested in hardware, especially digital electronics. I had the opportunity to get familiar with FPGAs in a couple of courses and realised that they should somehow be a part of my thesis. Having prior experience in software development and being a long time user of open-source software I was curious to investigate what the hardware community had to offer to create a complete system based on open-source components which could run Linux. The major part of the work was done at my employer's (Realway AB) office in Malmoe during the period April 2008 to September 2008.

I would like to thank Realway AB for sponsoring my master's thesis, Per Andersson at LTH for undertaking my master's thesis and giving me directions and feedback on the report, Daniel Mattsson at Realway AB for answering questions and giving feedback on the report and my opponent Alexander Jakobsen for giving me valuable feedback and suggestions on how to improve the report. Finally, I would also like to thank Anna Norling (my life partner) for supporting me throughout my studies which made it possible having a part-time job while attending university.

Mario Boikov
Helsingborg, September 2008

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Definition of open-source hardware . . . 1
- 1.3 Motivation . . . 2
- 1.4 Objectives . . . 3
- 1.5 Description . . . 3
- 1.6 Method . . . 3
- 1.7 Limitations . . . 5
- 1.8 Outline . . . 5

2 Theory **7**
- 2.1 Linux . . . 7
- 2.2 FPGA . . . 8

3 Analysis **11**
- 3.1 Introduction . . . 11
  - 3.1.1 LatticeMico32 . . . 11
  - 3.1.2 LEON3 . . . 11
  - 3.1.3 OpenRISC 1200 . . . 12
- 3.2 Hardware features . . . 12
- 3.3 License . . . 13
  - 3.3.1 LatticeMico32 . . . 14
  - 3.3.2 LEON3 . . . 14
  - 3.3.3 OpenRISC 1200 . . . 14
  - 3.3.4 Summary . . . 14
- 3.4 Documentation . . . 15
  - 3.4.1 LatticeMico32 . . . 15
  - 3.4.2 LEON3 . . . 16
  - 3.4.3 OpenRISC 1200 . . . 17
  - 3.4.4 Summary . . . 18
- 3.5 Peripherals . . . 18
  - 3.5.1 LatticeMico32 . . . 18

3.5.2 LEON3 . . . 18
3.5.3 OpenRISC 1200 . . . 19
3.5.4 Summary . . . 19
3.6 Development environment . . . 21
3.6.1 LatticeMico32 . . . 21
3.6.2 LEON3 . . . 23
3.6.3 OpenRISC 1200 . . . 26
3.7 Basic SoC . . . 26
3.8 Linux support . . . 27
3.8.1 LatticeMico32 . . . 27
3.8.2 LEON3 . . . 27
3.8.3 OpenRISC 1200 . . . 28
3.8.4 Summary . . . 28
3.9 Portability . . . 28
3.9.1 LatticeMico32 . . . 29
3.9.2 LEON3 . . . 29
3.9.3 OpenRISC 1200 . . . 29
3.9.4 Summary . . . 29
3.10 Analysis summary . . . 29

4 Implementation **33**
4.1 Available hardware . . . 33
4.2 Development environment . . . 34
4.3 Project plan . . . 34
4.4 The Linux kernel's input system . . . 36
4.4.1 Investigation . . . 36
4.4.2 Simple input device driver . . . 37
4.5 Creating the soft-core . . . 37
4.5.1 Debounce circuit . . . 37
4.5.2 Dual edge detector . . . 39
4.5.3 Button pad device . . . 40
4.6 Integrating with GRLIB . . . 41
4.7 Creating the device driver . . . 43
4.8 Acceptance test application . . . 44

5 Conclusions **45**
5.1 Conclusions . . . 45
5.2 Discussion . . . 46

6 Related works **47**

Abbreviations **49**

References **51**

List of tables **57**

List of figures **58**

Appendix **58**

A Bus specifications **59**
A.1 AMBA Rev 2.0 59
A.2 WISHBONE Rev B.3 60

Chapter 1

Introduction

In this chapter the background, objectives and method are presented.

1.1 Background

Linux in embedded systems is quite common today. Linux can be found in many different applications such as network storage units [31, 7], car navigation systems [5, 44] and media players [29], the list can be made long.

System-on-Chip (SoC) and embedded systems goes hand in hand. Field programmable gate arrays (FPGA) can be favorable[3] when building SoCs because most of the components and bus-communication can be placed inside the FPGA. Another advantage is that the FPGA is reconfigurable, new components can be added and old can be replaced or removed. The latter is also beneficial when doing bug fixes. A bug fix can be made in one day and do not require a "respin" as with application-specific integrated circuits (ASIC) which may require 2 to 6 months[3].

Deploying Linux on FPGAs is supported by major vendors such as Altera[9] (NIOS II) and Xilinx[58] (Microblaze) but not all of them releases the soft-cores under an open-source license.

This master's thesis will evaluate/investigate some of the available options to run Linux on an open-source hardware platform in terms of an FPGA-based SoC. It will also describe how to extend one of the SoCs and deploy Linux on it.

1.2 Definition of open-source hardware

Definition of the term *open-source hardware* by Edy Ferreira [21]:

> Any piece of hardware whose manufacturing information is distributed using a license that provides specific rights to users without the need to pay royalties to the original developers. These rights include freedom to use the hardware for

> any purpose, freedom to study and modify the design, and freedom to redistribute copies of either the original or modified manufacturing information.
>
> This definition fits what McNamara calls "open implementation" hardware, described as "hardware for which the complete bill of materials necessary to construct the device is available."
>
> In the case of open source software (OSS), the information that is shared is software code.In OSH, what is shared is hardware manufacturing information, such as hardware definition language descriptions, and the diagrams and schematics that describe a piece of hardware.

This master's thesis focuses on the availability of the hardware description language (HDL) source code.

1.3 Motivation

I belive that there are a number of advantages of using open-source hardware. Open-source software has proven successful and great examples are Linux, Firefox and Apache webserver. The European Commission services has in fact created a working group on libre software to analyse the effects of using open-source software[56]. The group released a paper[55] that contains, among other things, advantages of using open-source software. I belive that the advantages stated in the paper can be applied to open-source hardware too. I have extracted some of the advantages which are presented below. They have been modified with focus on hardware instead of software and motivates the usage of open-source hardware.

- *There is no single entity on which the future of the hardware depends.*
  No single entity can stop development of an IP core. If a company decides to drop support for a proprietary IP core no one has the right to continue the development. This can not happen with open-source hardware, it will even prevent monopoly because no company can buy the exclusive rights to an IP core which is released under an open-source license.

- *The availability of the source code and the right to modify it.*
  This enables tuning and improvement of the hardware. The availability of the source code makes it possible to port it to new technologies. It is also easier to isolate bugs and fix them when having access to the source code.

- *Possibility of "forking"*
  A good example is the Linux kernel. The Linux kernel depends on existence of a specific hardware unit (see the theory section 2.1). Without this unit the Linux kernel can not be used as is. A fork of Linux emerged that supports running Linux on hardware without this unit. This would not be possible if the source code was closed.

This can also be applied to a soft-core processor which might not have support for caches or other units and the authors are not interested in including the features. The project can be forked and maintained in parallel to the originated code.

- *Visibility*
  A thorough audit can be performed on the code to verify correctness. It also gives a detailed understanding of the architecture and operation.

1.4 Objectives

The objectives of this master's thesis is to evaluate SoCs based on three soft-core processors, see 1.5, and deploy Linux on one of them. The chosen SoC will be further extended with a custom made device that can be accessed from within Linux. A Linux device driver will be written to enable usage of the custom made device.

All soft-core components used in a SoC must be licensed under an open-source license, no proprietary soft-cores may be used.

1.5 Description

Three soft-core processors, as base for creating a SoC, has been identified: LatticeMico32[15], LEON3[2] and OpenRISC 1200[39]. They are all released under an open-source license. What makes the soft-cores interesting, other than the licensing, are the following properties that they all share:

- Toolchains based on GNU's not unix (GNU) compiler collection (GCC).
- Integrates with well known bus specification.
- Instruction set simulator.
- SoC support in terms of available components ready to be connected.
- Linux support.

The work consist of comparing the solutions and use one of them to implement a running Linux system.

1.6 Method

To achieve the objectives the work is divided into two phases, *analysis* and *implementation*. The phases are described in detail below:

- **Analysis**
  The *analysis* phase consist of doing a review and evaluation on each of the three soft-core processors. The purpose of this phase is to learn about the soft-cores, document features and collect comparison data. To structure and narrow the evaluation the following parameters are considered to be of importance for the comparison:

  - Hardware features - Features/options included in the soft-core processor.
    This will help identifying the differences between the three soft-core processors.
  - License - What license the soft-core is released under.
    This will help understanding how the solutions can be used.
  - Available peripherals - Supported peripherals.
    This will give an idea of what can be accomplished with a solution.
  - Documentation - Available documentation.
    The documentation can be a great help depending on the quality. This will give a hint of available documentation and the quality.
  - Development environment - Available tools.
    This will help identifying available tools provided and their usage.
  - Basic SoC - Complexity of assembling a basic SoC (processor, on-chip memory and serial port).
    This will give an idea of how easy/hard it is to assemble a SoC.
  - Linux support - What Linux versions are supported.
    This will identify the supported Linux version each solution provides.
  - Portability - The portability of the soft-cores.
    A portable solution gives a greater freedom to choose FPGA vendor and chip.

  The comparison data should serve as basis for choosing the soft-core processor to be used in the *implementation* phase.

- **Implementation**
  The implementation phase consist of building a SoC based on the solution chosen in the *analysis* phase. The SoC will be extended with a custom made soft-core component. This include writing a Linux device driver to enable usage of the device from within Linux, typically a device which reads buttons found on a development board.

  An application which utilizes the device should be written as an acceptance test.

  The intention with this phase is to explore to what extent the platform can be used. The outcome is a Linux system running on the SoC with the custom made device.

1.7 Limitations

The purpose of this master's thesis is not to port the Linux kernel to a new architecture, no attempts will be made to create a Linux port or modify the kernel source to support the SoC. Only a device driver will be written in the implementation phase.

Performance and resource usage of the SoCs will not be considered since a deployed soft-core can vary between different FPGA technologies, an FPGA can be designed for speed, area or power[37]. Another consideration is that a soft-core can itself be optimized for speed or area, for example using caches will gain speed to a cost of increasing area consumption. If any figures are presented in this report about performance or resource usage, they should only be considered as a hint and not as a comparable metric between the SoCs.

1.8 Outline

This report is organised as following:

- Chapter 1 gives an introduction to this master's thesis. A motivation of using open-source hardware is given, the objectives is formulated and the method is presented.
- Chapter 2 briefly discusses theory about Linux and FPGAs. The Linux part covers the basics about the Linux kernel, kernels modules and running Linux on hardware without memory management unit (MMU). The FPGA part gives an overview of the structure of an FPGA and the steps involved creating a downloadable bitstream. Readers with base knowledge about Linux and FPGAs can skip this chapter.
- Chapter 3 covers the *analysis* phase which includes details about the review and evaluation of the three solutions based on the parameters specified in 1.6. The author gives a comparision between each of the solutions.
- Chapter 4 presents a project plan for the *implementation* phase and describes the work involved implementing the solution.
- Chapter 5 presents the conclusions that emerged during this master's thesis. The author ends the chapter with a discussion about using open-source hardware and software.

In addition to the regular chapter the appendix contains an introduction to the Advanced Microcontroller Bus Architecture (AMBA) and the WISHBONE bus.

Chapter 2

Theory

The intention with the theory section is to give a brief introduction to Linux and FPGAs. The theory section consists of two sub sections and the first section explains what Linux is and the purpose of Linux kernel modules. The second section gives a basic overview of FPGA technology.

The content in the theory section is partially based on information from Wikipedia[53] [54] and the book "FPGA prototyping by VHDL examples"[6].

2.1 Linux

```
   Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big
and professional like gnu) for 386(486) AT clones.  This has
been brewing since april, and is starting to get ready.
I'd like any feedback on things people like/dislike in minix, as
my OS resembles it somewhat (same physical layout of the
file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem
to work. This implies that I'll get something practical within a
few months, and I'd like to know what features most people would
want. Any suggestions are welcome, but I won't promise I'll
implement them :-)

                Linus (torvalds@kruuna.helsinki.fi)

PS.  Yes - it's free of any minix code, and it has a multi-threaded
fs. It is NOT protable (uses 386 task switching etc), and it
probably never will support anything other than AT-harddisks,
as that's all I have :-(.
```

[50]

This post was made by Linus Torvalds on 25 August 1991. The text was posted on the usenet group *comp.os.minix* and started the community around the Linux kernel. Since then, the community has grown large and Linux version 2.6.24 had 1,057 developers contributing to the kernel [35].

Linux is an operating system which scales very well. It supports a large number of architectures and can be installed on workstations, servers and embedded systems. Linux is a monolithic kernel which means that the whole kernel is compiled into one large executable. One advantage is that the kernel can execute faster but the disadvantage is that the kernel needs to be re-compiled each time new functionality is to be added. To overcome this limitation the concept of kernel modules was introduced.

Linux kernel modules is used the extend to kernel with new functionality. A kernel module is a piece of code that can be loaded/unloaded while the operating system is running. This gives the benefit of adding functionality without needing to reboot the system nor re-compiling the kernel. A typical kernel module is a device driver which enables a hardware device to be used from within Linux. Actually, there is no restriction on what a kernel module can do, for example, a module can implement a filesystem or a library which expose functions that other modules can use.

Since Linux depends on virtual memory, and virtual memory requires an MMU, it is not suitable for microprocessors without an MMU. If a SoC is missing an MMU, microcontroller linux (uClinux) can be used instead of Linux. uClinux is a fork of Linux to enable Linux on hardware which does not support an MMU. One major disadvantage with uClinux is the lack of memory protection between processes, which can lead to crashes since all processes has access to each others memory area. Since version 2.5.45 of the Linux kernel most parts of uClinux has been integrated into the main line kernel.

2.2 FPGA

An FPGA is a device which can be configured to perform logical functions such as *AND*, *OR*, *XOR* or more complex combinatorial functions such as an *adder*. The typical FPGA consists of logic cells and programmable switches as shown in Figure 2.1.

A logic cell can be configured to perform a function and the switch handles the interconnection between the logic cells. In general a logic cell consists of an *n*-input look-up table (LUT) and a D flip-flop. A 4-input conceptual logic cell is shown in Figure 2.2. The LUT can be seen as a $2^n - by - 1$ memory and by writing its memory content properly the LUT can perform any *n*-input combinatorial function.

Modern FPGAs may also include higher level functionality into the silicon to reduce area and increase speed compared to build them from primitives. Multipliers, digital signal processing blocks, processors and memory are typical extensions included in an FPGA.

To configure an FPGA a bitstream need to be generated. Some kind of HDL is normally used to define the behaviour of the FPGA. Very high speed integrated circuit HDL (VHDL) and Verilog are two common HDLs. The steps to generate the bitstream, which is illustrated

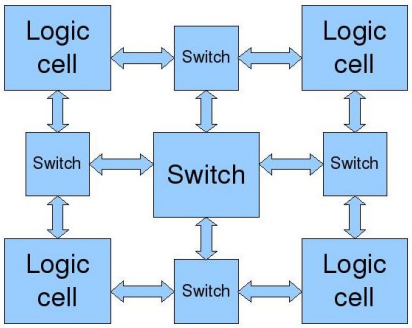

Figure 2.1: FPGA structure

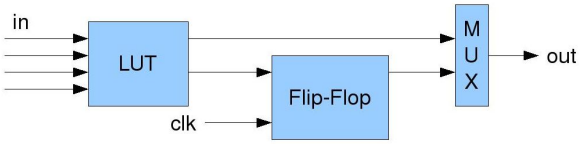

Figure 2.2: Conceptual 4-input logic block

in Figure 2.3, usually involves writing HDL, synthesis, implementation and finally generate the bitsteam which can be downloaded to the FPGA. The steps are described in more detail

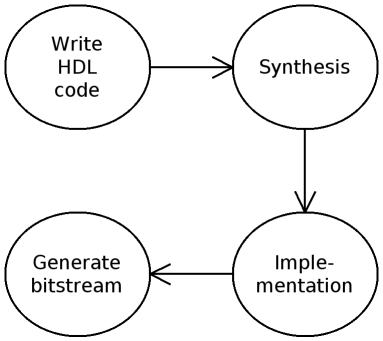

Figure 2.3: Generating bitstream

below:

1. Writing HDL - design and write HDL.
2. Synthesis - construct HDL to generic gate level components.
3. Implementation - consist of three smaller processes:
   - Translate - merges multiple design files to a signal netlist.
   - Map - maps the generic gates in the netlist to logic cells.

- Place and route - places the cell in physical locations and applies routing.

4. Generate bitstream - generate the bitstream from the final netlist and download to the FPGA.

One step that is not mentioned and not required is simulating. After writing HDL, it is quite common to simulate the HDL. The simulation is performed without the need for hardware and is a great way to verify functionality at an early stage.

Chapter 3

Analysis

In this chapter the three soft-core processors are reviewed and evaluated with respect to the parameters specified in 1.6.

3.1 Introduction

In this section an introduction to each of the three soft-core processors is given.

3.1.1 LatticeMico32

The LatticeMico32 soft-core is maintained by Lattice semiconductor corporation. Lattice designs, develops and markets a range of semiconductor components called programmable logic products. The company's product portfolio includes intellectual property (IP) cores, programmable logic devices, development kits and design software. The LatticeMico32 is a 32-bit Harvard reduced instruction set computer (RISC) processor and is written in Verilog.

3.1.2 LEON3

Gaisler Research is a company that provides IP cores and development tools for embedded processors based on the scalable processor architecture (SPARC). One of the products is the LEON synthesizable processor. The LEON3 is a 32-bit processor written in VHDL and compliant with the SPARC V8 architecture. The LEON3 is highly parametrizable through the use of VHDL generics and makes it possible to instantiate multiple processor cores in the same design with different configurations. The LEON3 is distributed as a part of the GRLIB IP library. The GRLIB IP library is a set of reusable IP cores designed for SoC development. LEON3 is certified by SPARC international as being SPARC V8 conformant.

3.1.3 OpenRISC 1200

OpenRISC 1200 is a 32-bit RISC processor implementation of the OpenRISC 1000 specification written in Verilog. OpenRISC 1000 is a popular open-source project hosted on the opencores.org website and OpenRISC 1200 is a sub-project. The project goal is to create a free, open-source computing platform which must be versatile to fit a wide range of applications. The OpenRISC 1000 architecture is a 32/64-bit load and store RISC architecture and targets medium and high performance networking, embedded, automotive and portable computer environments. In addition to OpenRISC 1200, the OpenRISC 1000 project hosts a couple of sub-projects such as SoC, simulator, Linux ports and toolchain.

3.2 Hardware features

All three soft-cores are 32-bit Harvard, RISC architecture processors and have separate buses and caches for instruction and data. Most of the features are optional and can be switched on and off before synthesis. They all have options to include a debug unit and multiply and divide instructions. Table 3.1 shows a comparison between the soft-cores.

Table 3.1: Comparison of hardware features

	LatticeMico32	**LEON3**	**OpenRISC 1200**
Pipline	6-stages	7-stages	5-stages
Multiply and accumulate (MAC)	No	Yes	Yes
SMP[a] support	No	Yes	No
System bus	WISHBONE	AMBA	WISHBONE
Power management unit	No	Yes[b]	Yes[c]
External interrupts	32	15	32
MMU	No	Yes	Yes
Floating-Point Unit (FPU)	No	Yes[d]	No
# Registers	32	40-520[e]	32

[a] Symmetric multi-processor
[b] Power-down and clock gating, see [26].
[c] Idle, doze, sleep-mode, see [38].
[d] Only available with commercial license.
[e] The SPARC uses register windows, see [30] and [26]

I was interested in how much FGPA resources each solution would occupy. I searched webpages and documents and found some figures in [13], [39] and [26]. The findings can be seen in Table 3.2. The figures should only be considered as a hint since the soft-cores are deployed on different FPGA technologies and I have not verified the correctness.

Table 3.2: FPGA real estate usage

	Resources	FGPA technology
LatticeMico32[a]	2158 LUTs	LatticeECP2/M
LEON3[b]	4300 LUTs	Virtex2
OpenRISC 1200[c]	3000 slices[d]	Virtex[e]

[a] Multiplier, Piped lined shifter, 8K I/D-caches
[b] 8K I/D-caches, other options are unknown
[c] No caches or MMUs
[d] Unknown number of LUTs
[e] The exact Virtex model was not specified.

The LatticeMico32 seems to be a quiet small processor, it only occupies 2158 LUTs and yet includes caches and multipliers. The number of LUTs for the OpenRISC 1200 can not be determined since it depends on which Virtex is used, for example Virtex 2 and Virtex 5 have different component architecture which makes the comparison difficult[33]. It is also possible that the tool spreads out the logic which will give an increased slice count.

Let's assume that they used a Virtex2 when synthesised the OpenRISC 1200, then the solution may occupy approximately 4000-6000 LUTs since each slice in Virtex2 contains two LUTs and all LUTs might not been utilised.

The LEON3 and OpenRISC 1200 does not seem to be suitable for small (less than 10k LUTs) FPGA devices. It will be difficult to fit a SoC when the processor occupies almost half of the available resources. Running Linux on the SoC will certainly at least require a memory controller, a timer, a UART and/or a VGA controller, and yet we do not have any kind of input device to the system such as a PS/2-keyboard controller. Add an ethernet MAC and a debug unit and I am sure this will exceed 10k LUTs. Estimating a SoC using figures from [26] will occupy 11450 LUTs (including the processor with mul/div and AHB/APB-bus) with the previously mentioned components.

My conclusion is that LatticeMico32 is targeted for small resource-contraint embedded systems. It does not have support for MMU, FPU, power managmenent and does not consume much resources. LEON3, which is SPARC V8 comformant, is more of a desktop/server processor with alot of features. The SPARC processor is typically found in computers from Sun Microsystems[45]. It seems that OpenRISC 1200 is competing with LEON3 by looking at the features list.

3.3 License

This section presents the licenses covering the soft-cores.

3.3.1 LatticeMico32

The LatticeMico32 is licensed under an open IP core license[14]. The license specifies that the generated HDL code is free to use and any proprietary designs can still remain proprietary. This allows implementation and distribution of hardware without a need for a separate license agreement. The license is a Lattice specific license.

The LatticeMico32 license is interesting, the license states that the *generated* HDL code is free to use. If I interpret the license correctly it says that only the generated HDL is available as open-source. This implies that you have to use the tools provided from Lattice to create a SoC because the tools contains the HDL templates for the IP cores. At the same time, the generated HDL code can be used as base and modified to ones' needs. Another interesting observation is that the generated HDL code contains a copyright note that states that the software is confidential and proprietary, this feels contradicting if the HDL is released as open-source.

3.3.2 LEON3

The LEON3/GRLIB is licensed under the GNU general public license (GPL) version 2[24] which requires that any changes to the source code or derived work need to be released under the same license. This means that if you put your own IP cores into the solution, those IP cores must also be released under the GPL. LEON3/GRLIB is also available under a commercial license which allows it to be used without the GNU GPL restrictions.

3.3.3 OpenRISC 1200

The OpenRISC project is licensed under the GNU lesser GPL (LGPL) version 2.1[24] which makes it possible to incorporate it into a proprietary design without having to release the proprietary code. Modifying the OpenRISC source code requires that the modifications must be released under LGPL.

3.3.4 Summary

What license is best is always an interesting question, it depends on who you ask. Some people prefer that the code is released under GPL, this ensures that the code is open and any work based on the code must also be released under GPL.

Personally I favour to base work on code that is LGPL because then I have the freedom to choose if I would like to release my code (not the LGPL code) as closed or open-source. I still have to release any changes to the LGPL'd code and include a copy of the license along with my solution which I think is good. This prevents the LGPL'd code from being exploited, any enhancements can be incorporated into the original code.

It seems that the Lattice license permits use of the code without any restrictions. I do miss a document, or at least I could not find one, about the Lattice license. I think that clarification around the license would be good. I am not sure how I can use the generated HDL more than

that I can base my work on it without having to release any proprietary HDL. For example, may I re-license it under GPL? I am not sure.

3.4 Documentation

In this section the evaluation of the documentation is presented. Only documentation that is found relevant is evaluated. Relevant documentation is counted as documentation about the processor, constructing SoCs, software development, integrating custom made components and Linux support.

3.4.1 LatticeMico32

Lattice has a large collection of documents for LatticeMico32, regarding everything from installation of tools to deployment of Linux. The homepage[15] contains a lot of information and is easy to navigate. A sub set of the documents are reviewed below.

- **LatticeMico32 Processor Reference Manual**[11]
  This manual covers details about the processor and its architecture. It includes sections about configuring the core and the WISHBONE interconnection architecture. The instruction set is also described in detail.

- **LatticeMico32 Software Developer User Guide**[12]
  This guide describes the process of creating, debugging and deploying software for the LatticeMico32 microprocessor. The guide describes LatticeMico32 System's run-time environment and the device driver framework which is used by the run-time environment. A section about the *Managed Build Process* gives an insight into the managed build process which is needed if any user-defined component is to be added. An *Advanced Programming Topic* section contains information about software deployment, the linker and memory sections, real time operating system (RTOS) support and how to create standard non managed build makefiles. Finally, a list with tools and their usage is presented.

- **Creating Components in LatticeMico32 System**[10]
  This guide describes how to add a custom component into Mico System Builder (MSB). The guide describes the process of creating new or edit existing components. The user interface is described in detail.

- **Linux Port to LatticeMico32 System Reference Guide**[16]
  This reference guide covers all details about the uClinux port for LatticeMico32. The guide describes how to customise the bootloader which initialises the board and provides information about the board to the Linux kernel. It also gives an outline of minimum requirements to be able to boot the Linux kernel. The section *Linux Port to*

LatticeMico32 System contains a lot of details about the Linux port. It also contains information about how to configure, building and deploying the kernel and userland application.

3.4.2 LEON3

Gaisler provides a sufficient set of documents. Some of the documents are from third party sources such as the AMBA and SPARC manual. The documents can be downloaded from the homepage[1].

- **SPARC v8 manual**[30]
  This manual covers details about the SPARC v8 architecture. No section covers processor configuration or integration with components due to that the document is not written by Gaisler Resarch. This manual seems to be a reference manual to implement a SPARC v8 processor, everything is written in fine-grained detail.

  A down-side with the manual is the absence of a table of contents. Topic of interest have to be found by searching.

- **AMBA-2.0 Manual**[40]
  This manual covers detail about the AMBA bus specification. This document is not produced by Gaisler Research therefore no information about Gaisler Research specific details are described in this manual. The manual starts with an introduction to AMBA and the three buses: AHB, Advanced System Bus (ASB), and APB. After the introduction an explanation of AMBA signals and naming convention is given which is followed by a detailed description of the three previously mentioned buses.

- **GRLIB IP Core User's Manual**[26]
  This manual covers the ip cores available in GRLIB. The introduction presents a lists with available IP cores containing the name, function, vendor/device id and license. The introduction continues with implementation characteristics by presenting a table with some of the IP cores and their approximate area usage on two different FPGAs and ASIC. The rest of the document describes each IP core in detail.

- **GRLIB IP Library User's Manual**[25]
  This manual covers GRLIB and how to extend GRLIB with new components. The introduction presents the concepts in GRLIB such as the library organisation, interrupts and plug and play capabilities. A quick start guide is given on how to implement a system using GRLIB. A section covers the installing details and what tools are supported in terms of synthesis and simulators. A section is given on how synthesis and simulations is performed. A sections explains the GRLIB design concepts covering the AHP and APB which is followed by a section presenting a design example. The last section in the document describes how to extend GRLIB with new IP cores.

- **LEON Integrated Development Environment for Eclipse**[22]
  This guide describes the installation process and basic usage of the Eclipse environment. A detailed description including screenshots are provided for the Windows installation process. The guide gives a short explanation of different concepts in Eclipse and gives a tour of where to find options and how to create projects and files.

- **SnapGear Linux for LEON**[28]
  This manual covers the details about running Linux and uClinux on LEON. The manual starts with giving an introduction to Linux for LEON and what IP cores are supported. The manual continues with explaining which toolchain to install depending on configuration and if uClinux or Linux will be used. This is followed by a section that describes in detail all possible configuration options for Linux, the hardware and user space applications. The manual also covers topics such as building software, adding custom application and debugging applications and the kernel. The last part of the document describes different configurations and installations.

3.4.3 OpenRISC 1200

The OpenRISC 1000 projects does not host many documents, the webpages functions as source of information.

- **OpenRISC 1000 Architecture Manual**[47]
  This document defines the OpenRISC architecture. The manual covers the instruction set, register set, cache management and coherency, memory model, exception model, addressing modes, operands conventions, and the application binary interface (ABI).

- **OpenRISC 1200 IP Core Specification**[38]
  This document covers the OpenRISC 1200 implementation. The specification describes the implementation specific variables that are not part of [47]. It covers details about the type and size of data and instruction caches, type and size of data and instruction MMUs, details of pipelines, implementation of exception unit, interrupt controller and other supplemental units.

- **Basic Custom OpenRISC System Hardware Tutorial**[49]
  Two third party tutorials are provided by the Embedded Systems Group at De Nayer Instituut[32] in Belgium, one for Altera devices and one for Xilinx devices. The tutorial helps composing and implementing a OpenRISC SoC with minimum number of peripherals, a debug-unit, an on-chip memory and a UART.

- **Basic Custom OpenRISC system Software Tutorial**[48]
  This third party tutorial is also provided by the Embedded Systems Group at De Nayer Instituut[32] in Belgium. The tutorial describes how to build and install the toolchain, create a simple “hello world” application which can be downloaded using GDB and a JTAG-cable.

3.4.4 Summary

The documentation from Gaisler Research could be a bit more descriptive. The technical details such as the IP cores were described well. I was especially interested in documentation of how to extend the SoC since that is one of the tasks in the implementation phase. The section about extending GRLIB with a new IP core was very techical and contained a lot of details. The down-side with the section is the lack of examples. The only example that exists is quite obvious, just connecting the AMBA, and plug&play signals. I think Gaisler Research should include an example with a step-by-step guide on how to add a new IP core to GRLIB that is not already AMBA-ready. I have the same opinion about the *Integrated Development Environment for Eclipse* document, once again I think Gaisler Research should try design the document as a step-by-step guide using scenarios instead of just listing options on how to do different things in Eclipse, but that is my personal opinion.

Overall I found the documentation from Lattice satisfying. The documentation was easy to read and contained step-by-step guides which I prefer.

The OpenRISC project provides only documentation for the architecture and the OpenRISC 1200 processor. Both documents explained their respective topics sufficient. The lack of other documentation is a major down-side for the OpenRISC project.

3.5 Peripherals

This section presents available peripherals each solution provides. All IP cores for each solution presented in this section are assumed to be compatible with their respectively soft-core processor.

3.5.1 LatticeMico32

Lattice offers commonly used peripherals, such as general purpose input/output (GPIO), timer and universal asynchronous receiver/transmitter (UART), under the open IP core license. Some cores requires a license purchased from Lattice, like the double data rate (DDR) memory controller. The LatticeMico32 is WISHBONE compliant and can be integrated with other cores which uses the same open bus specification. A large set of WISHBONE compliant open-source cores can be found at opencores.org. Table 3.3 presents available open-source IP cores that are included in the installation package.

3.5.2 LEON3

Gaisler Research provides a wide range of peripherals through the GRLIB IP library. Common IP cores such as UART, timer, GPIO and memory controller can be found in the library. The peripherals are based around the AMBA bus standard. The GRLIB IP library is released under the GNU GPL. As with the LEON3 core, GRLIB IP library can also be licensed under

Table 3.3: Available IP cores for LatticeMico32

Name	Function	License
LatticeMico32	Soft-core processor	Lattice open IP core
GPIO	DMA controller	Lattice open IP core
UART	RS-232 serial communication	Lattice open IP core
SPI	Serial peripheral interface	Lattice open IP core
I2C Master	I2C master from opencores	Public domain[a]
Timer	Timer unit	Lattice open IP core
SDRAM Controller	SDRAM memory controller	Lattice open IP core
SPI Flash ROM	SPI flash ROM controller	Lattice open IP core
Parallel flash	Parallel flash controller	Lattice open IP core
On-chip memory	On chip memory unit	Lattice open IP core
Async SRAM	Asyncronious SRAM controller	Lattice open IP core

[a]Copyright/license message must be preserved in source code.

a commercial license. Table 3.4 presents available open-source IP cores found in GRLIB. The table is not exhaustive, more IP cores are available, see [26].

3.5.3 OpenRISC 1200

The OpenRISC 1000 project itself does not maintain any peripherals. The project do provide a reference SoC platform that can be used as starting point. The SoC includes devices such as UART, ethernet MAC and simple VGA interface. Since the OpenRISC is WISHBONE compliant, it can be integrated with other cores which uses the same open bus specification. A large set of WISHBONE compliant open-source cores can be found at opencores.org. Table 3.5 presents the IP cores included in the reference SoC.

3.5.4 Summary

Both LatticeMico32 and LEON3 includes a wide range of peripherals which makes the two solutions more interesting than OpenRISC 1200. The OpenRISC 1000 project does not provide any peripherals (except from the ones included in a reference SoC), which I consider as a major down-side. Altough OpenRISC 1200 is WISHBONE compliant, the cores on opencores.org are not verified to work with OpenRISC 1200. There is no guarantee that an IP core that have a WISHBONE interface will work flawless with the soft-core processor.

LEON3, or more correctly GRLIB, includes a large amount of IP-cores. Gaisler Research has put a lot of effort to make the cores technology independent which makes them usable on FPGAs from different vendors. Each core has a set of signals in addition to the AHB/APB bus signals to add plug&play support, this includes automatic address decoding, interrupt

Table 3.4: Available IP cores in GRLIB

Name	Function	License
LEON3	Soft-core processor	GPL
AHBRAM	Single-port RAM with AHB interface	GPL
AHBROM	ROM generator with AHB interface	GPL
GRETH	Gaisler Research 10/100 Mbit Ethernet with AHB I/F	GPL
SPICTRL	SPI Controller with APB[a] interface	GPL
I2C	I2C Master and slave with APB interface	GPL
APBUART	Programmable UART with APB interface	GPL
APBPS2	PS2 Keyboard interface with APB interface	GPL
CAN_OC	Opencores CAN 2.0 MAC with AHB interface	GPL
PCITARGET	32-bit target-only PCI interface	GPL
PCIARB	PCI Bus arbiter	LGPL
AHBCTRL	AMBA AHB bus controller with plug and play	GPL
APBCTRL	AMBA APB Bridge with plug and play	GPL
DDRCTRL	8/16/32/64-bit DDR controller with two AHB ports[b]	GPL
MCTRL	8/16/32-bit PROM/SRAM/SDRAM controller	LGPL
SDCTRL	PC133 SDRAM controller	GPL
SVGACTRL	Video graphics array (VGA) controller core	GPL
GPTIMER	General purpose timer	GPL
GRGPIO	Scalable general-purpose I/O port	GPL

[a] Advanced Peripheral Bus
[b] Xilinx only

steering and device identification. The LEON3 soft-core itself does not use the plug&play capabilities, it is only the AHB/APB controllers which uses the information. The devices can be enumerated from software by reading appropriate addresses on the AMBA bus. The AHB/APB controllers in GRLIB has been extended with distributed address decoding, this implies that the bus does not need to be modified each time a core is added or removed. This is not the case with the other solutions.

The down-side with the LatticeMico32 is that two important cores requires a permanent purchased license, the DDR memory controller and the ethernet MAC. A free limited time evaluation license exist for this two cores. I could not find any information on how these two IP cores may be used in a design. No effort has been made by Lattice to make their IP cores technology independent, for example block ram primitives are instantiated in the HDL code.

Table 3.5: IP cores included in the OpenRISC 1000 reference SoC

Name	Function	License
OpenRISC 1200	Soft-core processor	LGPL
Ethmac	Ethernet MAC	LGPL
Debug interface	Development debug interface	LGPL
PS2	Mouse and keyboard PS/2 controller	LGPL
SSVGA	Simple small VGA	LGPL
uart16550	UART 16550 compatible device	LGPL

3.6 Development environment

This section presents the evaluation of the provided tools for each of the solutions. The evaluation focuses on tools that helps constructing SoCs and writing software. All the software is installed and executed on Kubuntu 8.04[36] (Linux distribution).

3.6.1 LatticeMico32

Lattice supports both a toolchain and an integrated development environment (IDE). The LatticeMico32 system can be downloaded from the LatticeMico32's homepage[15]. The package includes the Eclipse based IDE, a java run-time required to start Eclipse, GCC version 3.4.4 for LatticeMico32, soft-core components, template platforms and a couple of utilities. No registration is required to download the package.

Toolchain

GCC is used as toolchain and provides following features:

- GNU C/C++ cross compiler.
- GNU debugger (gdb).
- Small C-library.
- Instruction set simulator.

The instruction set simulator is a part of the GNU simulator/debugger framework. The simulator is not cycle-accurate, it executes one instruction per cycle and does not model pipeline delays. The simulator supports only one peripheral as default, a serial port or UART memory mapped at 0x80000000. The source code for the simulator has to be modified and re-compiled to include support for any other device.

The LatticeMico32 system

LatticeMico32 system is the name of the development environment and is based on the Eclipse C/C++ development tools (CDT) environment. It contains two integrated tools to help building embedded systems, MSB and the C/C++ software project environment (SPE). Following features can be found in MSB:

- Generate platform description and associated HDL for hardware implementation.
- Choose peripheral components to attach to the LatticeMico32.
- Specify connectivity between peripheral components.

SPE provides following features:

- Develop the code that runs on platforms created with MSB.
- Interfaces via command line to compiler, assembler, linker, and debugger tools.
- Target the Instruction set simulator or hardware.

Developers familiar with Eclipse will recognise the environment that consist of Eclipse with a couple of plug-ins. Wizards exists for creating SoCs and Mico32 C/C++ applications.

When working in the MSB perspective, the developer can connect different peripherals to the WISHBONE bus as can be seen in Figure 3.1. Relevant parameters for each peripheral

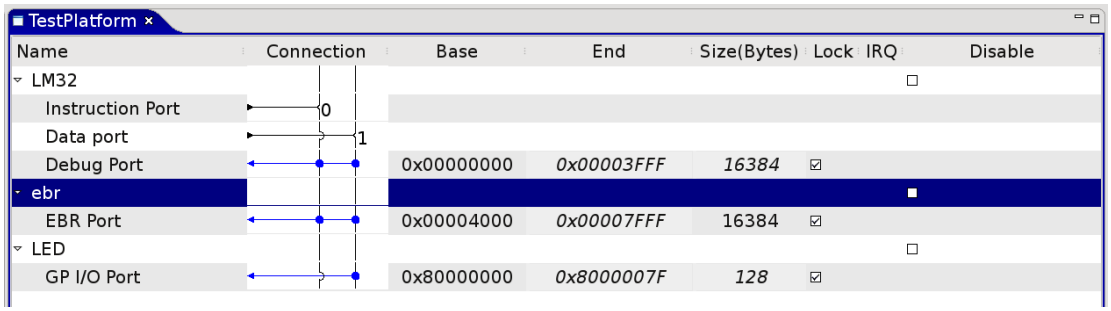

Figure 3.1: Connecting peripherals in MSB

can be edited by double-clicking on the peripheral. Figure 3.2 shows an example of editing UART properties.

Peripherals are added to the SoC by double-clicking on a component from a list of available components. The instance appears in the same view as shown in Figure3.1 and appropriate connections can be made to the WISHBONE bus.

Creating software for a SoC is done by executing one of the Mico32 C/C++ application wizards. The wizard associates a SoC with the software project which makes it easy to access the peripherals on the SoC from the application. The application can be debugged in the

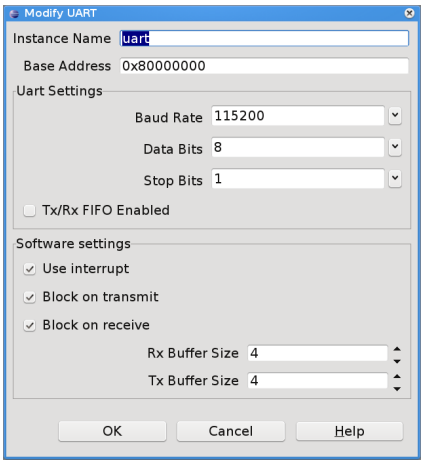

Figure 3.2: Editing UART properties

instruction set simulator that is included in the LatticeMico32 system. By default only serial port is supported as peripheral in the simulator. The source code to the simulator needs to be modified to support other peripherals.

There are also options to download bitstream and software to a Lattice development board.

3.6.2 LEON3

Gaisler Research provides a rich set of tools to aid development. The tools can be downloaded from the homepage[1]. The tools are packaged separately and are released under different licenses. An installer with all tools are supported for the Microsoft Windows platform. No registration is required to download the package.

Graphical SoC configuration

LEON3 can be configured by using a graphical tool built on tkconfig from the Linux kernel. The configuration tool presents available options and peripherals and an example is shown in Figure 3.3. The build system is based on *makefiles* and the graphical configuration tool is started from command-line by executing *make xconfig*. GRLIB contains pre-defined designs for a set of development boards which can be used as base when doing configuration. Each design declares menu options for which peripherals is supported for the board which is used when using the graphical configuration. If no pre-defined design exists one have to be created.

LEON Bare-C Cross Compiler System

BCC is a GNU based cross compiler used for C/C++ compilation and debugging. BCC has following features:

- GNU C/C++ cross compiler.

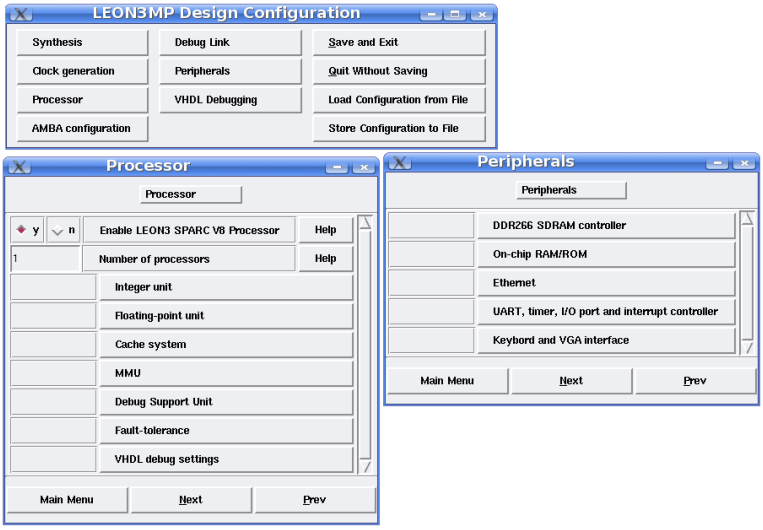

Figure 3.3: LEON configuration

- GNU debugger (gdb).
- Newlib embedded C-library.
- Pthreads library.
- Bare-C run-time system with interrupt support and tasking.
- Boot-programmable read-only memory (PROM) builder.

GRMON

Grmon is a debug monitor which communicates with the debug support unit (DSU) which can be found in LEON processors. Some of the features of grmon:

- Read/write access to all registers and memory.
- Downloading and execution of applications.
- Remote connection to GNU debugger.
- Serial, ethernet, JTAG, PCI and universal serial bus (USB) debug interfaces.

A graphical version of grmon is also available, it is based on the Eclipse rich-client platform and is named grmon-rcp. Grmon is not distributed for free. An evaluation version can be downloaded to try out the tool.

GRSIM/TSIM

Grsim/tsim are simulators of the LEON processor. Tsim emulates a mono-processor system and grsim emulates a multi-processor system. The grsim is time-based and tsim instruction-based. Both simulators can emulate I/O functions. The simulators supports a basic set of peripherals from GRLIB as default. The simulator can be extended with loadable modules which enabled user defines input/output devices. None of the simulators are distributed for free, an evaluation version can be downloaded to try them.

Eclipse IDE

Eclipse can be used as IDE by installing a plug-in that is provided by Gaisler Research for free with full source code. To install the plug-in, the software update feature of Eclipse can be used. The plug-in enables cross compiling and debugging from within Eclipse.

The PATH environment variable need to be modified to point to the installation of the toolchain, else the plug-in is not able to find the toolchain and there is no configuration options to point to the toolchain directory. This is not the case for the simulators and grmon where it is possible to select the binary by browsing the filesystem from a file dialog.

To create a new software project the C/C++ wizard is executed and one of the SPARC toolchains are selected which is shown in Figure 3.4. The plug-in provides facilities to debug an application (integrates with the debug perspective) from within Eclipse either by using the simulator or grmon.

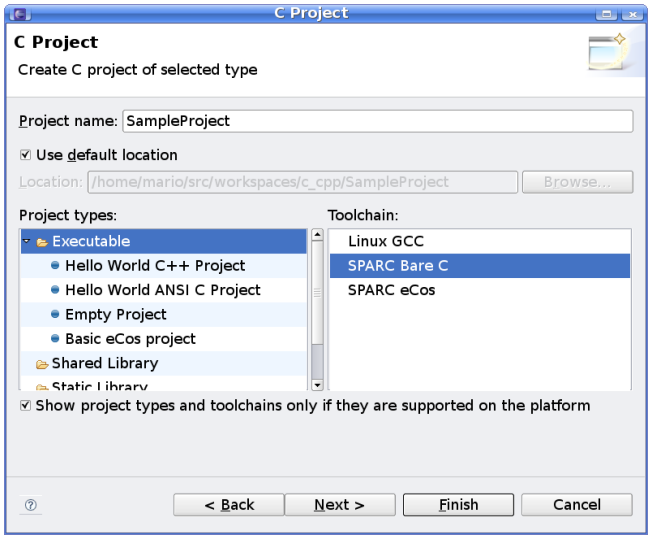

Figure 3.4: C/C++ wizard in Eclipse (LEON3)

3.6.3 OpenRISC 1200

The OpenRISC project provides both a toolchain and an architectural simulator. The toolchain need to be compiled since no binaries are provided.

GNU toolchain port

The GNU toolchains port is an OpenRISC port of the GNU based cross compiler. It is used for C/C++ compilation and debugging. The port has following features:

- GNU C/C++ cross compiler.
- GNU debugger (gdb).
- Newlib embedded C-library.

Currently, a port for the 64-bit RISC does not yet exist.

Architectural Simulator

The Architectural simulator is an OpenRISC 1000 architecture simulator which can emulate OpenRISC based computer systems. The simulator provides following features;

- All major models of OpenCores peripheral and system controller cores supported.
- Easy addition of new peripheral models.
- Remote debugging through a network socket with the GNU debugger.
- Support for different environments (memory configurations and sizes, OpenRISC 1000 processor model, configuration of peripheral devices).

Currently, only the 32-bit RISC is supported.

3.7 Basic SoC

This section presents my attempt to create a basic SoC for each of the processors. The work is based on the findings from section 3.6.

Each SoC should contain a processor, an on-chip memory (which could contain the application if the SoC was deployed on a FPGA) and a UART to be able to output “Hello World”. The SoCs should be executed in a simulator.

Creating the SoC for LatticeMico32 respectively LEON3 was easy, it took a couple of minutes to configure the SoCs by using the configuration tools from each provider. My opinion is that LatticeMico32 provides a slightly better environment. It gives a better overview of your SoC. You can visually see the connection of the peripherals to the WISHBONE bus

and are able to modify addresses and interrupts for the peripherals. LEON3 only provides a solution where you can choose what peripherals should be included and does not visualise the SoC. On the other hand, GRLIB supports plug&play so managing the connections and the ability to set addresses and interrupts are pointless. It is also possible that I might have been influenced by the fact that everything is done in Eclipse for the LatticeMico32 solution, everything from adding componentes to the SoC to writing the "Hello World" application and start it in the simulator. The LEON3 SoC must be configured outside Eclipse.

OpenRISC 1200 is not that easy to deal with. If you only have the IP cores for the SoC available without an interconnection component you will have to write a bus arbiter (a bus arbiter grants a component access to the bus) yourself. The easiest way to create a SoC is to base the design on the reference SoC. This is very error-prone and takes time since adding a component means that you need to write the component instantiation and then connect the component to the bus manually.

3.8 Linux support

In this section I will present the supported Linux solution provided for each soft-core processor.

3.8.1 LatticeMico32

The LatticeMico32 is MMU-less and can only run uClinux. Theobroma-Systems[43] maintains a uClinux 2.0 port including toolchains which is used to compile Linux application binaries. The Linux kernel and toolchains can be downloaded from the LatticeMico32 uClinux webpage[42] hosted at Theobroma-Systems. It seems that the distribution is based on the SnapGear distribution because the main window's title is "SnapGear Embedded Linux Configuration". A rich set of applications are available and can be configured to be included.

All Lattice IP cores from the installation package are supported in uClinux.

3.8.2 LEON3

Linux support for LEON is based on the SnapGear distribution and can be downloaded from Gaisler Research's homepage[1]. Two Linux versions are supported, the uClinux 2.0 series for non-MMU systems and the Linux 2.6 series for MMU systems. It is also possible to choose between two C-runtimes, glibc and uClibc. Glibc is normally used in Linux systems and uClibc is optimised and targeted for embedded systems. A rich set of applications are available and can be configured to be included.

According to [28], at least 14 IP cores are supported in Linux.

3.8.3 OpenRISC 1200

The OpenRISC project maintains ports for both Linux and uClinux. The Linux version is based on the 2.4-kernel series and uClinux is based on the 2.0 release. The kernel sources can be downloaded from [19] and [20]. The project also tries to maintain device drivers for various open cores peripherals.

3.8.4 Summary

Both LEON3 and LatticeMico32 uses a Linux distribution from SnapGear[17]. The distribution makes it very easy to compile the kernel and user-space applications. The LatticeMico32 biggest disadvantage is that it is only capable of running uClinux. This might not be sufficient when high reliability (memory protection) is preferred to avoid system crashes, for example in medical applications. Trying to run a uClinux system with the LatticeMico32 simulator could not be accomplished since the simulator needs to be extended with basic peripherals to boot a simple kernel, the provided simulator contains only a UART.

I tried to run Linux with TSIM (the LEON3 simulator) but ran into problems. The kernel booted but after a while it stopped and nothing happened. After some time an error message was printed: “Simulation time overflow”. I decided to try GRSIM instead, and it worked. It did stop at the same location as with TSIM but continued after a while and finally the shell prompt emerged. I tried a pre-compiled uClinux kernel from the GRSIM package with TSIM and it worked. Maybe TSIM is only capable of running uCLinux, it had the same problem with another pre-compiled Linux (full Linux) kernel from the GRSIM package. GRSIM had no problems running any of the kernels.

I checked out Linux and uClinux from the OpenRISC version control repository. The OpenRISC project seems to base the uClinux on the release from the uClinux project and a stock Linux 2.4 kernel with OpenRISC patches. User-space application need to be compiled and added manually. The uClinux directory contains a readme file with instructions on how to compile the tools, kernel, simulator and how to run the kernel with the simulator. I managed to compile the 2.4 kernel but not the uClinux kernel. The build stopped and complained about “junk at the end of line” in the file head.S. I did not succeed to start the 2.4 kernel in the architectural simulator, an unhandled exception occurred. I did not try to solve the problems since I think that the provided settings should be sufficent to try running the kernels with the supplied simulator config.

3.9 Portability

In this section the investigation of the portability of the SoCs are presented.

3.9.1 LatticeMico32

LatticeMico32 utilises technology dependant primitives so the solution can only be deployed on Lattice FPGAs. Some attempts has been made to port the processor with success according to this forum post[23]. The port mentioned in the forum post only covers the processor and not the peripheral IP cores.

3.9.2 LEON3

GRLIB, which includes LEON3, is highly portable. The IP core library supports FPGAs from Actel, Altera, Lattice and Xilinx. A complete list with supported development board (template designs) can be found on Gaisler Research's homepage[1].

3.9.3 OpenRISC 1200

OpenRISC 1200 is fairly portable, according to [39] the soft-core is portable to Xilinx and Altera devices.

3.9.4 Summary

GRLIB is really an impressive collection of reusable and portable IP cores. The list of supported development boards contains 25 entries with four different FPGA vendors. None of the other two solutions come close to that. LatticeMico32 is only supported on Lattice devices and OpenRISC 1200 should at least work on devices from Xilinx and Altera.

I have only measured the portability based on what is provided by the vendors/projects, it does not mean that the solutions can not run on other devices than enumerated in this section. The soft-cores are released with an open-source license which makes it possible to port the core to other FPGA technologies. This is one of the motivations of using open-source soft-cores, becoming technology independent.

3.10 Analysis summary

The statements in this section is the author's own opinions and the reader is free to agree or disagree with them.

My opinion is that the OpenRISC 1000 project seems to be a bit chaotic. When I first visited the webpages I got the impression that the project was mature and no major problems should occur using the solution. I got a bit suspicious when I checked the timestamps on files in the version control repository. The timestamps showed that most of the files has not been touched in two to six years. I started to inspect the webpages in more detail and it did not take long until I noticed instructions where they stated that the toolchain has only been tested under Red Hat 7 (Linux distribution), I think Red Hat 7 was released in the year of 2000 and is

obsolete now. I had severe problems compiling the debugger and had to install Red Hat Linux 7 (by using virtualization software) to succeed compiling the debugger. The project could at least host statically linked binaries on the webpage. Another example is the Linux page which states that a 2.6 kernel is about to be added into the version control repository, nothing has happend for almost three months (the time I have monitored the page). Most things does not works "out of the box" and require manual tweaking. The simulator failed to run the Linux kernel as described in 3.8.4 and uClinux did not compile.

On 17 July 2008, approximately half of the time writing this master's thesis, the development seems to take off and a VMWare image with a complete updated SoC development environment is released. This will hopefully have the effect of soft-cores being updated and new tools introduced to assemble SoCs. It can be worth monitoring this project to see how it evolves. I did not do an evaluation on this new release due to time constraints. I did try to start the image in VMWare and everything worked and I could start a pre-compiled Linux 2.6 kernel in the provided simulator, this looks promising.

LatticeMico32 is an interesting solution because it seems to be fast and not very resource hungry. The major down-side with LatticeMico32 is that the solution requires a separate commercial license for the DDR memory controller and ethernet MAC. IP cores from opencores.org may be used to remove this hinder. Only having support for uClinux can also be considered as a down-side compared to LEON3 and OpenRISC, but this depends on the application. I really liked the SoC composition environment in Eclipse and the step-by-step guides.

I believe that LEON3/GRLIB is the technical superior solution of these three solutions. It has separate buses for low-speed and high-speed devices (AHB and APB) which should yield higher performance since a UART can be located on the low-speed bus while main memory is located on the high-speed bus. The number of included IP cores is large and technology independent. Although the SoC composition is not integrated with Eclipse, configuration of the SoC and IP cores are still easy using the tkconfig graphical interface. Running software on LEON3 without actual hardware was simple using the provided simulators. None of the other solutions were as good as GRSIM/TSIM. The default LatticeMico32 simulator could only simulate a serial port, the source code needs to be downloaded and modified to extend the simulator. GRISM/TSIM could easily be extended without modifying the simulator code. Actually, the OpenRISC simulator was not that bad but GRSIM/TSIM felt more robust and mature.

Table 3.6 shows a comparision with grades between 1-5, where 1 is low and 5 is high. The comparision is based on the evaluation and conclusions from this chapter. The SoC from Gaisler Research was chosen for the *implementation* phase. GRLIB contain a lot more IP-cores than the other solutions, it is more portable and is capable of running both Linux and uClinux. LatticeMico32 can only run uClinux and OpenRISC is not as mature as GRLIB.

Table 3.6: Graded comparision of the SoCs

	LatticeMico32	LEON3	OpenRISC 1200
Hardware	3	5	4
Documentation	5	3	2
Peripherals	3	5	1
Development environment	5	4	1
Basic SoC	5	5	2
Linux support	3	5	3
Portability	2	5	3
Avarage	3.7	4.6	2.3

Chapter 4

Implementation

This chapter presents details and work involved during the implementation of the Linux device driver and the soft-core which enables reading buttons found on the development board.

4.1 Available hardware

An XUP-V2Pro development board is available for the *implementation* phase. The development board is sold by Digilent, Inc. and more detailed information is available at Digilent's homepage[18].

Some of the features that can be found on the development board are following:

- Virtex-2 Pro XC2VP30 FPGA.
- Up to 2Gb DDR synchronous dynamic random access memory (SDRAM).
- 10/100 Ethernet port.
- 100 MHz system clock oscillator.
- Compact Flash card slot.
- AC-97 audio Codec.
- Serial advanced technology attachment (SATA), super extended graphics array (SXGA), personal system/2 (PS/2) and recommended standard 232 (RS-232) ports.

The Virtex-2 Pro is a FPGA from Xilinx. It contains 30,816 logic cells and should be large enough to house a SoC with a couple of peripherals. The board was chosen because of three reasons, it contains a fairly large FPGA, the board is one of the supported boards in GRLIB and Xilinx provides free development tools (WebPack)[57] supporting both Microsoft Windows and Linux desktops. There is no technology reason why a Xilinx FPGA is chosen.

The development board also includes five push-buttons which are shown in Figure 4.1. This buttons will be a part of the custom made device.

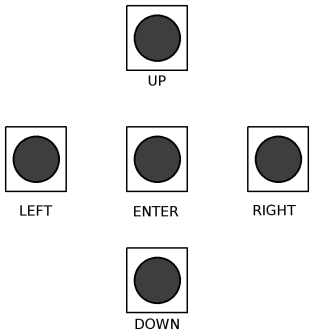

Figure 4.1: Buttons on the development board

4.2 Development environment

- Xilinx WebPack 9.2 is used as development environment for the hardware. The hardware implementation is done using VHDL.
- The software is developed using GCC, a text editor (kate from K Desktop Environment) and makefiles. C is used as programming language.

4.3 Project plan

The project was divided into small iterations with clear goals. Each iteration was designed to focus on only one topic at a time. The following iterations were identified:

1. **Investigating the Linux kernel's input system**
   The purpose of this iteration is to get familiar with the input system in Linux. Two action points are identified:

   (a) Find and study material about the input system.

   (b) Build a sample input device.

   The goal of the first action point is to learn about the input system and what is required to create an input device driver.

   The goal of the second action point is to get hands-on experience of the input system. The sample device driver should only generate a couple of key events which the input system should interpret as real key events from a device. To accomplish this task, some kind of timer may be used to trigger key events from within the device driver. If a solution based on a timer is used, this action point also involves finding and study material about timers.

   The source code can be reused when implementing the real device driver.

2. **Creating the soft-core.**
   The purpose of this iteration is to create the soft-core which generates an interrupt as soon as a button is pressed or released. It should also be possible to read button values.

   This iteration comprises of the following action points:

   (a) Build a debounce circuit.

   (b) Build a dual edge detection circuit.

   (c) Build the button pad device using the debounce and dual edge detection circuit.

   The purpose of the debounce circuit is to prevent glitches in the signal when a button is pressed or released, this may occur since the button is a mechanical device[6].

   The dual edge detection circuit should sense when a button is pressed or released and generate a *tick* each time this occur.

   The purpose of the last action point is to assemble the circuits to a complete soft-core.

   Testbenches should be created to verify the functionality of the soft-core.

3. **Integrating with GRLIB.**
   The purpose of this iteration is to learn about the AMBA bus specification and how to integrate the button pad soft-core with GRLIB. The following two action points are identified:

   (a) Find and study material about the AMBA bus and GRLIB.

   (b) Create an AMBA wrapper for the soft-core from the previous iteration.

   The goal of the first action point is to learn about the AMBA bus and GRLIB. This is required to be able to extend the LEON3 with custom made soft-cores.

   The purpose of the second action point is to creates the AMBA wrapper. This also includes adding the soft-core to GRLIB as a re-usable component.

4. **Creating the device driver.**
   In this iteration the Linux input device driver is created. The device driver should be interrupt based and not poll the button pad device. To accomplish this iteration following action points is identified:

   (a) Find and study material about how to use interrupts in Linux.

   (b) Create the button pad device driver.

   The goal of the first action point is to learn about how interrupt-programming is done in Linux.

   The purpose of the second action point is to extend the device driver written in the first iteration to support the button pad soft-core.

5. **Acceptance test application.**
   The goal of this final iteration is to write an acceptance test application which demonstrates the button pad device driver. The application should be a command line application which prints a message each time a button is pressed.

4.4 The Linux kernel's input system

4.4.1 Investigation

I started to search information about the input system using Google, but quickly I came to the insight that I could actually look in the source code instead of getting my information from third party sources. I downloaded and extracted Linux kernel 2.6.25.9 from [34]. It did not take too much time for me to find a document[1] that coveres the basics of creating a simple input device driver. Two other files was mentioned in that document, *input.c* and *input.h*. These two files are located in *drivers/input* and *include/linux*. I found that these two files are a great source of information and does not only contain the source code but also documentation of functions and structures.

The input system seems to be pretty easy to use and supports a wide range of devices like mouses, keyboards and joysticks. It can also handle output events which goes from the system to the input device. Such devices can be the PC-speaker, keyboards with status leds or joysticks supporting force feedback. A feature with the input system is that it handles key autorepeat and does not rely on hardware support. I think that this is an advantage because key autorepeat can be supported for devices that does not support it in hardware and the driver does not need to know any details about it.

In Table 4.1 I present functions that has been identified as relevant to the button pad device driver implementation. More detailed information about the relevant functions and structures can be found in the files mentioned above.

Table 4.1: Relevant input system functions

Name	Description
`input_allocate_device()`	Allocates memory for a new input device.
`input_register_device()`	Register device with input system.
`input_unregister_device()`	Unregister previously registered device.
`input_report_key()`	Report new key event.

[1] Documentation/input/input-programming.txt

4.4.2 Simple input device driver

I could create a simple input device driver by using the information from the *investigation* action point. I was surprised how easy it was to create a simple input device, I had expected a little more work.

The solution included a timer-driven key event generator. I could not find any documentation about timers in the *Documentation* folder so I used [8] as reference.

A kernel timer is scheduled to execute a function at a particular time in the future. The time specified is based on *jiffies* and not an absolute date and time. *Jiffies* is a counter that is initialised to zero at boot and incremented each timer interrupt. The timer interrupt interval is programmed at boot time with an architecture-dependent value. The kernel export functions to convert between different time formats and the function prototypes can be found in *include/linux/jiffies.h*. The source code and headers for timers can be found in *kernel/timer.c* and *include/linux/timer.h*. For the simple input device driver I could identify two timer-functions and one time convert function that was relevant and they are listed in Table 4.2.

Table 4.2: Relevant timer-functions

Name	Description
`init_timer()`	Initialises a timer structure.
`add_timer()`	Start a timer.
`msecs_to_jiffies()`	Converts milliseconds to jiffies.

During this action point, I discovered that inspecting driver code is a great source of information, not only about the current topic, but also about what is provided from the kernel in terms of functions and macros. For example the macro *ARRAY_SIZE()* (which gives the number of elements in an array), found in the header file *include/linux/kernel.h*, saves a lot of typing. I do also believe it prevents the programmer from doing mistakes when iterating arrays because the *sizeof()* operand in C gives the total size in bytes of the array, not the number of entries.

The simple input device driver was build and tested on Kubuntu 8.04 which at the moment was running Linux kernel 2.6.24-19.

4.5 Creating the soft-core

4.5.1 Debounce circuit

The purpose of this component is to prevent glitches in the signal when pressing/releasing a button. Figure 4.2 shows a conceptual example of how a signal can look like when a button is pressed and released.

I created an outline 4.3 of the debounce circuit to get a general idea of the circuit. The circuit consist of four states. The first state (is_pressed) sets the output to zero which means

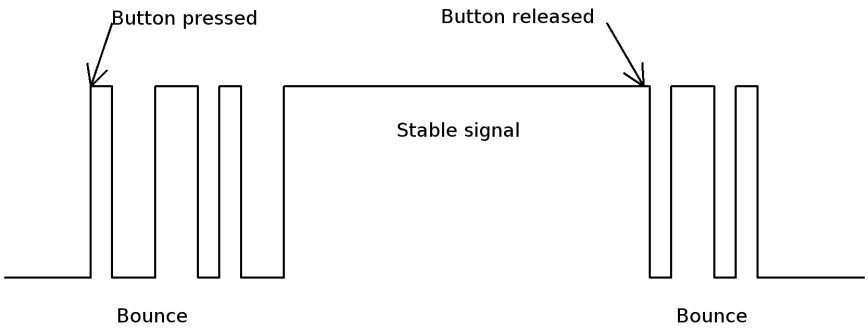

Figure 4.2: Signal with bounces

that the button has not been pressed and waits for the button to be pressed. When the button is pressed the circuit loads a counter with a value and enters the second state (debounce_down) which handles the press-bounces. If the button is pressed it keeps decreasing the counter until it reaches zero and then advances to the third state (is_released). If a bounce occurs it goes back to the first state and the process starts all over again. When the third state is entered the circuit signals that the button has been pressed by raising the *button_down* signal and keeps it high while the button is pressed down. When the button is released the circuit advances to the fourth state (debounce_up) which takes care of release-bounces. The fourth state works just like the second with the difference of checking if the button is released instead of being pressed. When the bounces are filtered out the circuit goes back to the first state which completes the iteration by lowering the *button_down* signal.

During the design of the circuit it crossed my mind that I could replace the debounce-circuit by implementing the logic in software. The device driver could take care of the bounces simply by checking if a specific time had expired between two consecutive events. If the button events differed the time was reset and the process of checking bounces restarted, else an event had occurred and could be reported to the input system. After some considerations I dismissed the idea because the processor will be interrupted a lot, once for each bounce and this prevents it from doing more important things. It will also be quite expensive in terms of context-switches[52] since each time an interrupt occurs the current running process will be halted and the interrupt routine takes up the execution. A poll-driven solution was also considered but that might end up being worse since the driver will have to poll the device at regular time intervals and not only on button down/up events. I came to the conclusion that it is better to implement the debounce-circuit in hardware instead of putting the load on the processor with a software solution.

Table 4.3 shows resources utilised after synthesis.

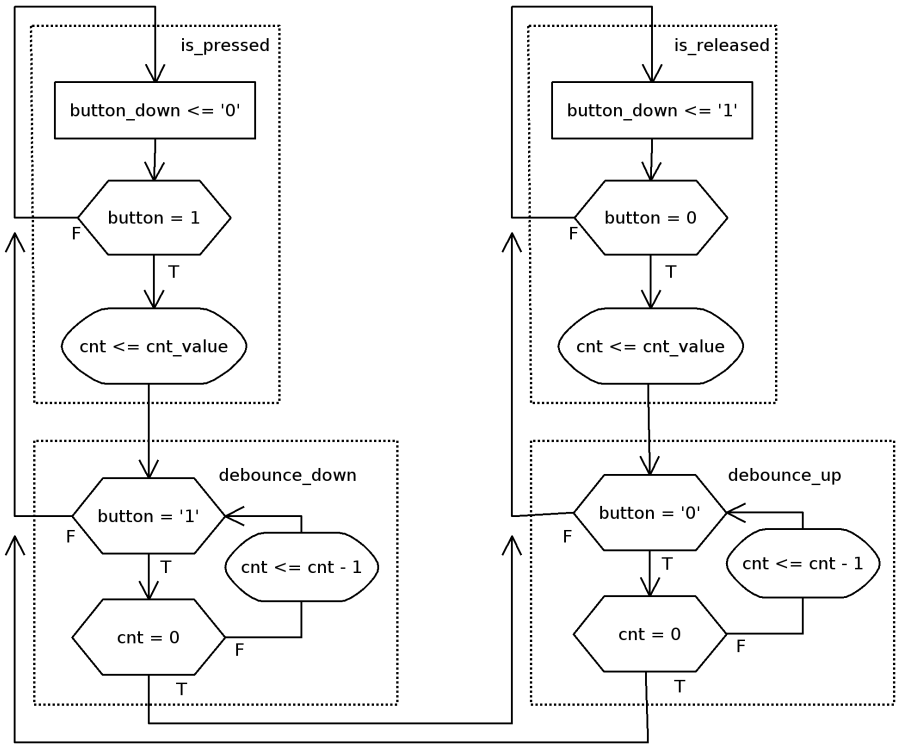

Figure 4.3: Debounce flowchart

4.5.2 Dual edge detector

The responsibility of the dual edge detector is to generate a tick signal when a button is pressed or released. Figure 4.4 illustrates the concept of the circuit. The dual edge detector is a simple

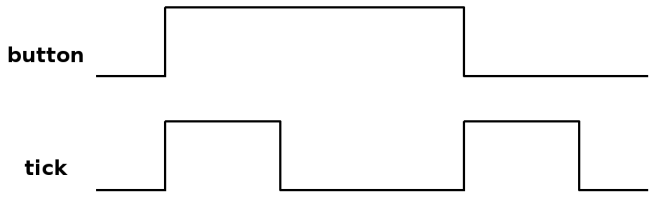

Figure 4.4: Dual edge detector concept

circuit that detects edge changes on the button signal and generates a tick for maximum one clock cycle each time the button signal goes from low to high or from high to low. The dual edge detector will be used as base for generating interrupt signals by the button pad device. Table 4.4 shows resources utilised after synthesis.

Table 4.3: Debounce resource utilisation

Logic Utilisation	Used	Available	Utilisation
Number of Slices:	27	13696	0%
Number of Slice Flip Flops:	23	27392	0%
Number of 4 input LUTs:	50	27392	0%

Table 4.4: Dual edge detector resource utilisation

Logic Utilisation	Used	Available	Utilisation
Number of Slices:	1	13696	0%
Number of Slice Flip Flops:	1	27392	0%
Number of 4 input LUTs:	1	27392	0%

4.5.3 Button pad device

The purpose of this entity is to assemble the two previously created circuits into one soft-core which will become the button pad device. Figure 4.5 shows an overview of the device. The button pad entity consist of *button* instances and interrupt logic. Each *button* entity manages one physical button and contains a *debounce* instance and a *dual edge detector* instance. The button pad component features `generics` and makes use of `generate` statements to support arbitrary number of buttons.

The button pad is responsible of generating interrupt signals when button events occur. It generates an interrupt signal by "sensing" button events from the *button* instances. The interrupt mechanism is depicted in Figure 4.6. When a button event occur, the button pad senses the event and raises the interrupt signal. The interrupt signal is held high until an acknowledge is received on the *int_ack* input. If no acknowledgement is received and a new edge change is detected, the interrupt signal will be held high and it is up to the entity that is using the circuit to notice this and take appropriate actions.

Table 4.5 shows resources utilised by a five-button circuit after synthesis.

Table 4.5: Button pad (5 buttons) resource utilisation

Logic Utilisation	Used	Available	Utilisation
Number of Slices:	143	13696	1%
Number of Slice Flip Flops:	126	27392	0%
Number of 4 input LUTs:	254	27392	0%

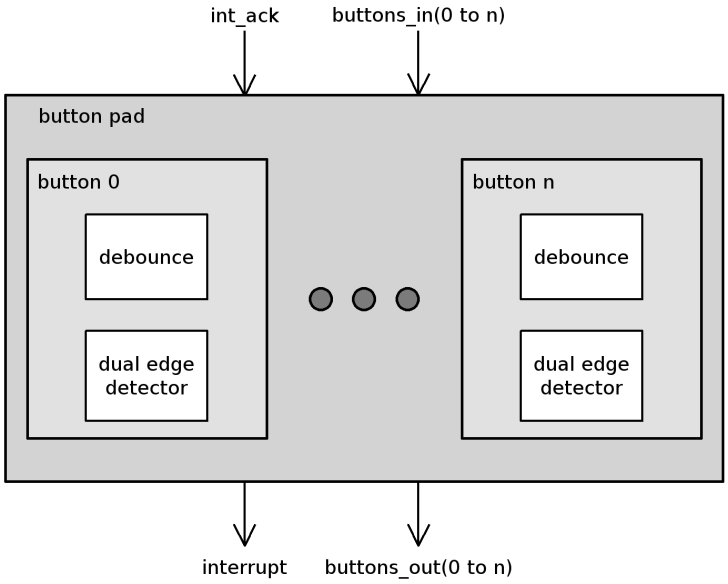

Figure 4.5: Overview of the button pad device

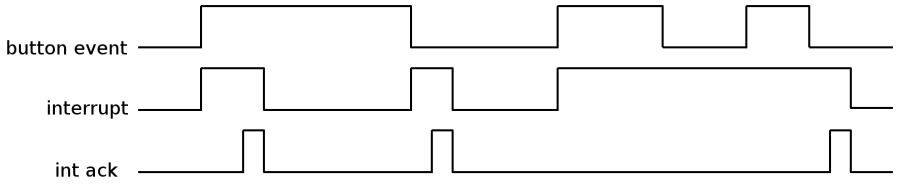

Figure 4.6: Interrupt mechanism

4.6 Integrating with GRLIB

To integrate the button pad device as a component in GRLIB, I came up with the following strategy:

1. Use GRLIB IP Library User's Manual[25] as source of information for extending GRLIB.

2. Try to find an implementation in GRLIB that resembles the button pad, such as an GPIO device which can be used as a reference.

[25] covered details about the directory layout for components in GRLIB and what files needs to be present and their contents.

I found two VHDL files which could be of used as reference when implementing the APB wrapper for my device. The first one is a GPIO implementation (*lib/openchip/gpio/apbgpio.vhd*) and the second is a PS/2 implementation (*lib/gaisler/misc/apbps2.vhd*).

While I studied the code for the two devices I noticed that there were no interrupt acknowledgement mechanism in the devices. After searching for more information in the documentation I found out that the interrupt controller handles the acknowledgement, this means that I can always acknowledge my button pad device by settings the interrupt acknowledge signal high.

I based my design on the *leon3-digilent-xup* design which is provided in GRLIB. Extending the design was not that difficult. I added my button pad device to the lib directory with the appropriate directory layout and added a couple of configuration files which are needed by the build system. The following files was modified to extend the design with my device.

- leon3mp.ucf - Mapping of net signals to I/O pads on the FPGA
- leon3mp.vhd - Adding an instantiation block for the button pad device
- config.in - Enable the button pad device in the configuration menu

I had problems synthesising the design, the tool complained about that the button pad entity did not exist. This happened because I had already synthesised the design before I extended the GRLIB with my button pad device. The build system cached the available component libraries so my button pad was not available to the build system. I did a cleanup and then I was able to synthesis the design. This was frustrating and a waste of time.

To test the integrated button pad device, I created a simple application that only read the values (polling) from the button pad. The device did not work, I only got zero-values. After spending a lot of time on this problem the button pad started to report values. The following things was changed to make the device report values:

- Using low voltage transistor transistor logic (LVTTL) level for the button pads. Without this, grmon could not find any devices on the AMBA bus.
- The push buttons on the development board is active low so I had to invert the signals before passing them to the button pad device. I should have tried this on hardware earlier and not only in simulation.
- GRLIB uses active low reset and the button pad uses active high, nothing happened in the button pad device since it was stuck in the reset-code. I chose to invert the reset signal inside the APB wrapper.

The changes was not sufficient, the buttons stopped to report values after using them for a while. This occurred randomly and I could not identify any patterns. To eliminate the risk of erroneous software, I verified by reading the values directly from within grmon and got the same result as with the software. I spent a lot of time trying to nail down why this happened. I suspected that the state machine for the debounce circuit could be the problem but after inspection and lot of testing (even on other development boards) I could not find any errors. I ignored the problem for the moment and started to implement an interrupt based version of the

simple test software. I found some example code in the software folder provided in GRLIB on how to add interrupt handlers. This worked at the first attempt but I still had the problem of "hanging" buttons. Suddenly, when I searching the Internet about *"debounce"* and *"push buttons"* I noticed the word *"metastability"* which can occur when you have asynchronous input. It became obvious for me that I needed to sample the buttons to a register before working with the values in the button pad circuit. After adding a register which is used to hold the button values the button pad started to work correctly. This also shows that things can work in simulation but not on real hardware.

Lesson learned from this action is to test the soft-core on hardware between iterations. For example, it would be a lot easier to find the problem of active low signals if I would have tested the debounce circuit on hardware, and not only by simulating the circuit.

4.7 Creating the device driver

The simple input device driver created in the first iteration was used as base to create the button pad device driver. I identified two things that needed to be done beside implementing the driver:

1. Find out how to access the IRQ and address for my device via the AMBA plug&play bus.

2. Integrate my button pad device driver to the SnapGear build system.

I could not find any documentation about how to read the IRQ and address, so I started to search in the Linux input sub-system since I suspected that there should be a driver for the PS/2 device which is included in GRLIB. A driver for the PS/2 device was found in *drivers/input/serio/leon3ps2.c* which I used as reference. The driver contained code on how to fetch device information on the AMBA bus. The implementation of the AMBA plug&play functions are located in *drivers/ambapp*.

Adding the module to the build system was trivial since the Linux kernel build system is very easy to use. I only had to add one entry in a "Kconfig" file and one in a makefile. No additional steps was required to integrate with SnapGear.

In the same source, the PS/2 driver, I could easily extract information on how to use interrupts in Linux. I did also read a chapter about the topic in [51] to learn more about interrupts in Linux. While inspecting the PS/2 driver code I noticed that all read and writes to the device was done using the two following macros, `LEON_BYPASS_LOAD_PA` and `LEON_BYPASS_STORE_PA`. It seems that this macros prevents the data from being cached by the processor.

The button pad driver generates keyboard input events when one of the buttons is pressed or released, the buttons are mapped as cursor keys. I was able to boot the kernel and see that my driver found my button pad device.

4.8 Acceptance test application

The acceptance test application is a trivial application, it only prints out which button is pressed. The difficult part in this application is to put the standard input into raw mode. This is done to prevent the operating system from buffering the input before the application receives it, instead each character is forwarded to the application directly. I searched the Internet for examples on how to enter raw mode and found a lot of them.

My goal was also to integrate the application into the SnapGear build environment. The file *"Documentation/Adding-User-Apps-HOWTO"*, which can be found in the Snapgear distribution, describes step-by-step what has to be done to add the application to the menu of available application and include it in the resulting filesystem.

Basically, the work involved integrating the application consisted of adding a couple of one-line entries in configuration and makefiles.

Chapter 5

Conclusions

This chapter presents and discusses the conclusions that have emerged during this master's thesis.

5.1 Conclusions

This master's thesis presents three options to run Linux on open hardware in terms of an FPGA-based SoC. Gaisler Research's solution is the most portable and have widest range of peripherals. Lattice's solution is the easiest to get started with but it lacks some hardware features such as an MMU which makes it only possible to run uClinux. It is also the solution which is least portable, for the moment only Lattice devices are supported altought the community is doing some progress in this field. The solution from the OpenRISC project is the hardest to configure and set up. No tools are available from the project to create a SoC and no development environment exists.

My work shows that FPGAs are suitable for creating SoCs, in my case I could extend a SoC with a button pad device without major effort. I did also show how easy it is to repair an error, for example the problem with the inverted reset only required a trivial change in the VHDL code, re-generating the bitstream and download it to the FPGA. This takes approximately 20-30 minutes (design uses approx. 16000 4-input LUTs) on my laptop equipped with an Intel Core2 CPU at 2.00GHz running Kubuntu 8.04.

The implementation part of this master's thesis shows that open hardware and open software can be used to create complete systems with custom made hardware and software. I managed to implement the system by myself without depending on support or well written documentation, only having access to the source code was sufficient. For example by studying the code for the GPIO-core I could fairly quickly understand how to integrate my core with the AMBA bus and supporting plug&play. I did also take advantage of existing source code when I created the Linux driver by studying code on how to find devices on the AMBA bus and at the same time I gained information about interrupts in Linux. This shows how powerful and efficient it can be to have access to source code.

5.2 Discussion

I find it very interesting to base solutions on FPGAs which gives a very flexible solution. Gaisler's solution supports instantiating one or more processors, this means that you can create systems with support for more than one processor on the same circuit board without modifications. Linux also has multi-processor support and the only difference between a single processor and multi-processor system would be the FPGA resources occupied. Most vendors have several size options for the same FPGA series which makes it possible to select the appropriate chip depending on the final product.

It is also possible to replace an open core with a proprietary, for example a JPEG codec for a camera. The AMBA wrapper could take care of translating operations, else the plug&play capabilities makes it easy to load the appropriate Linux driver and the application will not be aware of the change as long as the driver has the same interface towards the application, nothing has to be modified on the circuit board. Since Gaisler's solution is released under the GPL this is not permitted because all hardware must be released under a GPL compatible license. Gaisler offers a commercial licenses which makes it possible to incorporate proprietary code and still have access to the source code from Gaisler. Open-source does not have to be free of charge.

An interesting observation I did during this master's thesis is that the open hardware community is quite small compared to the open software community. Opencores.org seems to be the number one place for open hardware. Two of the three solutions I evaluated are maintained by companies and are not community-based. During this master's thesis I constantly looked for new solutions but could not find any, it seems that these three are the only available with Linux support. I hope this change so there can be a variety of solutions that fits in different situations and with different open-source licenses.

I did also realise that using FPGA-based SoCs with Linux is a great educational tool. The students gets the opportunity to create hardware, software and learn operating system concepts, as with my button pad device. The solution from Gaisler Research can be used which comes with the full HDL source code at no cost.

Chapter 6

Related works

This chapter enumerates related works found during my master's thesis.

- **Using an FPGA Processor Core and Embedded Linux for Senior Design Projects [27]**
  This paper describes the experiences of using FPGA-based SoCs with open-source RTOS for educational purposes. During the implementation I found that using FPGA and open-source operating systems is a great educational tool so I agree with the authors of the paper.

- **Building a SoC for industrial applications based on LEON microprocessor and a GNU/Linux distribution [4]**
  This paper was released during my master's thesis and shows that it is feasible to create SoCs based on open hardware and open software. The paper describes an implementation of a SoC based on LEON3 which is extended with a modified UART from OpenCores. The SoC runs Debian Linux.

  Much of the work in the paper resembles the work in my master's thesis.

Abbreviations

ABI	Application Binary Interface
AHB	Advanced High-speed Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASB	Advanced System Bus
ASIC	Application-Specific Integrated Circuit
DDR	Double Data Rate
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GPIO	General Purpose Input/Output
GPL	General Public Licence
HDL	Hardware Description Language
IDE	Integrated Development Environment
IP	Intellectual Property
JTAG	Joint Test Access Group
LGPL	Lesser General Public Licence
LUT	Look-Up Table
MAC	Multiply and ACcumulate/Media Access Control

MMU	Memory Management Unit
PCI	Peripheral Component Interconnect
PHY	PHYsical layer
PROM	Programmable Read-Only Memory
PS/2	Personal System/2
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RS-232	Recommended Standard 232
RTOS	Real Time Operating System
SATA	Serial Advanced Technology Attachment
SDRAM	Synchronous Dynamic RAM
SoC	System-on-Chip
SPARC	Scalable Processor ARChitecture
SXGA	Super eXtended Graphics Array
UART	Universal Asynchronous Receiver/Transmitter
uClinux	MicroController Linux
USB	Universal Serial Bus
VGA	Video Graphics Array
VHDL	Very High Speed Integrated Circuit HDL

References

[1] Gaisler Research AB. Gaisler research's homepage, 2008. [Online; accessed 8-July-2008]
http://www.gaisler.com/.

[2] Gaisler Research AB. Leon3's homepage, 2008. [Online; accessed 4-July-2008]
http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53.

[3] M. Abramovici, C. Stroud, and M. Emmert. Using embedded FPGAs for soc yield improvement. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 713–724, 2002.

[4] A.Muñoz, E.Ostúa, M.J.Bellido, A.Millán, J.Juan, and D.Guerrero. Building a soc for industrial applications based on leon microprocessor and a gnu/linux distribution. Technical report, Departamento de Tecnología Electrónica - Universidad de Sevilla, 2008. http://www.dte.us.es/id2/publicaciones/congresos/2008-07-isie-1.pdf.

[5] TomTom International BV. Tomtom's homepage, 2008. [Online; accessed 4-June-2008]
http://www.tomtom.com/page.php?Page=gpl.

[6] Pong P. Chu. *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*. Wiley-Interscience, Hoboken, N.J, Mar. 2008. Includes bibliographical references (p. 437-438) and index.

[7] Inc Cisco Systems. Nslu2's homepage, 2008. [Online; accessed 4-June-2008]
http://www.linksys.com/servlet/Satellite?c=L_Product_C2&childpagename=US%2FLayout&cid=1118334819312&pagename=Linksys%2FCommon%2FVisitorWrapper&lid=1931222279B58.

[8] Jonathan. Corbet, Alessandro. Rubini, and Greg. Kroah-Hartman. *Linux device drivers*. O'Reilly, Beijing, 3rd ed. edition, 2005. Includes bibliographical references (p. 575-577) and index.

[9] Altera Corporation. Altera's homepage, 2008. [Online; accessed 6-May-2008] http://www.altera.com.

[10] Lattice Semiconductor Corporation. *Creating Components in LatticeMico32 System*. Lattice Semiconductor Corporation, Sep. 2007. http://www.latticesemi.com/documents/doc21617x14.pdf.

[11] Lattice Semiconductor Corporation. *LatticeMico32 Processor Reference Manual*. Lattice Semiconductor Corporation, 6.1 sp1 edition, Sep. 2007. http://www.latticesemi.com/documents/doc20890x45.pdf.

[12] Lattice Semiconductor Corporation. *LatticeMico32 Software Developer User Guide*. Lattice Semiconductor Corporation, 6.1 sp1 edition, Sep. 2007. http://www.latticesemi.com/documents/doc20893x66.pdf.

[13] Lattice Semiconductor Corporation. Latticemico32 architecture, 2008. [Online; accessed 19-July-2008] http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/mico32architecture.cfm.

[14] Lattice Semiconductor Corporation. Latticemico32 open source licensing, 2008. [Online; accessed 17-July-2008] http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/mico32opensourcelicensing.cfm?source=sidebar.

[15] Lattice Semiconductor Corporation. Latticemico32's homepage, 2008. [Online; accessed 24-June-2008] http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/index.cfm.

[16] Lattice Semiconductor Corporation. *Linux Port to LatticeMico32 System Reference Guide*. Lattice Semiconductor Corporation, Feb. 2008. http://www.latticesemi.com/documents/lm32_linux_port_ref_guide.pdf.

[17] Secure Computing Corporation. Snapgear embedded linux distribution home page, 2008. [Online; accessed 18-July-2008] http://www.snapgear.org/index.html.

[18] Inc. Digilent. Digilent's homepage, 2008. [Online; accessed 19-June-2008] http://www.digilentinc.com/.

[19] Marcus Erlandsson. Opencores linux page, 2008. [Online; accessed 4-June-2008] http://www.opencores.org/projects.cgi/web/or1k/linux.

[20] Marcus Erlandsson. Opencores uclinux page, 2008. [Online; accessed 4-June-2008] http://www.opencores.org/projects.cgi/web/or1k/uclinux.

[21] Edy Ferreira. Open hardware business models, 2008. [Online; accessed 30-September-2008] http://www.osbr.ca/ojs/index.php/osbr/article/view/570/523.

[22] Jonas Flodén. *LEON Integrated Development Environment for Eclipse*. Gaisler Research, 1.3.0 edition, Feb 2008. http://gaisler.com/eclipse/lide/gr_lide.pdf.

[23] LatticeMico32 forum. How to use latticemico32 open source in nonlattice devices, 2008. [Online; accessed 09-July-2008] http://www.latticesemi.com/forums/forum/messageview.cfm?catid=522&threadid=10504.

[24] Inc. Free Software Foundation. Gnu general public license, version 2, 2008. [Online; accessed 17-July-2008] http://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

[25] Jiri Gaisler, Edvin Catovic, and Sandi Habinc. *GRLIB IP Library User's Manual*. Gaisler Research, 1.0.18 edition, 2007. http://www.gaisler.com/products/grlib/grlib.pdf.

[26] Jiri Gaisler, Edvin Catovic, Marko Isomäki, Kristoffer Glembo, and Sandi Habinc. *GRLIB IP Core User's Manual*. Gaisler Research, 1.0.18 edition, May 2008. http://www.gaisler.com/products/grlib/grip.pdf.

[27] T.S. Hall and J.O. Hamblen. Using an fpga processor core and embedded linux for senior design projects. Technical report, Southern Adventist Univ., Collegedale, 2007. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=4231436&isnumber=4231412.

[28] Daniel Hellström. *SnapGear Linux for LEON*. Gaisler Research, 1.36.0 edition, Dec 2007. ftp://gaisler.com/gaisler.com/linux/linux-2.6/snapgear/snapgear-manual-1.0.36.pdf.

[29] Popcorn Hour. Popcorn hour's homepage, 2008. [Online; accessed 4-June-2008] http://www.popcornhour.com.

[30] SPARC International Inc. *The SPARC Architecture Manual*. SPARC International Inc., sav080si9308 edition. http://www.gaisler.com/doc/sparcv8.pdf.

[31] Synology Inc. Synology's homepage, 2008. [Online; accessed 4-June-2008] http://www.synology.com.

[32] De Nayer Instituut. Embedded systems at de nayer instituut, 2008. [Online; accessed 13-July-2008]
http://emsys.denayer.wenk.be.

[33] National Instruments. Advantages of the xilinx virtex-5 fpga, 2008. [Online; accessed 17-July-2008]
http://zone.ni.com/devzone/cda/tut/p/id/7440.

[34] Kernel.org. The linux kernel archives, 2008. [Online; accessed 26-June-2008]
http://www.kernel.org/.

[35] Greg Kroah-Hartman. Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it, 2008. [Online; accessed 18-June-2008]
http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php.

[36] Kubuntu.org. Kubuntu homepage, 2008. [Online; accessed 26-June-2008]
http://www.kubuntu.org.

[37] Ian Kuon and Jonathan Rose. Area and delay trade-offs in the circuit and architecture design of fpgas. Technical report, The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, 2008. http://portal.acm.org/citation.cfm?id=1344671.1344695&coll=Portal&dl=ACM&CFID=76207640&CFTOKEN=13191131.

[38] Damjan Lampret. *OpenRISC 1200 IP Core Specification*. opencores.org, rev.0.7 edition, Sep. 2001. http://www.opencores.org/cvsget.cgi/or1k/or1200/doc/or1200_spec.pdf.

[39] Damjan Lampret. Openrisc1000's homepage, 2008. [Online; accessed 4-July-2008]
http://www.opencores.org/projects.cgi/web/or1k/openrisc_1200.

[40] ARM Limited. *AMBA specification*. ARM Limited, May 1999. http://www.gaisler.com/doc/amba.pdf.

[41] ARM Limited. Arm limited's homepage, 2008. [Online; accessed 14-July-2008]
http://www.arm.com.

[42] Theobroma Systems Limited. Latticemico32 uclinux webpage, 2008. [Online; accessed 24-June-2008]
http://www.theobroma-systems.com/mico32/.

[43] Theobroma Systems Limited. Theobroma systems' homepage, 2008. [Online; accessed 24-June-2008]
http://www.theobroma-systems.com.

[44] Garmin Ltd. Garmin developer, 2008. [Online; accessed 4-June-2008] http://developer.garmin.com/linux.

[45] Sun Microsystems. Sun microsystems' homepage, 2008. [Online; accessed 21-July-2008] http://www.sun.com.

[46] OpenCores. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. OpenCores, b.3 edition, Sep. 2002. http://www.opencores.org/projects.cgi/web/wishbone/wishbone.

[47] Opencores. *OpenRISC 1000 Architecture Manual*. opencores.org, April. 2006. http://www.opencores.org/cvsget.cgi/or1k/docs/openrisc_arch.pdf.

[48] Patrick Pelgrims, Tom Tierens, and Dries Driessens. *Basic Custom OpenRISC system Software Tutorial*. DE NAYER Instituut, 1.00 edition, 2003. http://emsys.denayer.wenk.be/?project=empro&page=cases&id=6.

[49] Patrick Pelgrims, Tom Tierens, and Dries Driessens. *Basic Custom OpenRISC System Hardware Tutorial*. DE NAYER Instituut, 1.00 edition, 2004. http://emsys.denayer.wenk.be/?project=empro&page=cases&id=6.

[50] Linux Torvalds. What would you like to see most in minix?, 2008. [Online; accessed 17-June-2008] http://groups.google.com/group/comp.os.minix/msg/b813d52cbc5a044b?dmode=source.

[51] Sreekrishnan Venkateswaran. *Essential Linux device drivers*. Prentice Hall open source software development series. Prentice Hall, Upper Saddle River, NJ, 2008. Includes index.

[52] Wikipedia. Context switch — wikipedia, the free encyclopedia, 2008. [Online; accessed 30-June-2008] http://en.wikipedia.org/w/index.php?title=Context_switch&oldid=211084099.

[53] Wikipedia. Linux kernel — wikipedia, the free encyclopedia, 2008. [Online; accessed 17-June-2008] http://en.wikipedia.org/w/index.php?title=Linux_kernel&oldid=219873873.

[54] Wikipedia. Uclinux — wikipedia, the free encyclopedia, 2008. [Online; accessed 17-June-2008] http://en.wikipedia.org/w/index.php?title=%CE%9CClinux&oldid=218494042.

[55] European working group on libre software. Free software / open source: Information society opportunities for europe?, 2000. [Online; accessed 18-July-2008] http://eu.conecta.it/paper/.

[56] European working group on libre software. European working group on libre software, 2008. [Online; accessed 18-July-2008] http://eu.conecta.it/.

[57] Inc. Xilinx. Ise webpack software, 2008. [Online; accessed 7-July-2008] http://www.xilinx.com/ise/logic_design_prod/webpack.htm.

[58] Inc. Xilinx. Xilinx's homepage, 2008. [Online; accessed 6-May-2008] http://www.xilinx.com.

List of Tables

3.1 Comparison of hardware features 12
3.2 FPGA real estate usage 13
3.3 Available IP cores for LatticeMico32 19
3.4 Available IP cores in GRLIB 20
3.5 IP cores included in the OpenRISC 1000 reference SoC 21
3.6 Graded comparision of the SoCs 31

4.1 Relevant input system functions 36
4.2 Relevant timer-functions 37
4.3 Debounce resource utilisation 40
4.4 Dual edge detector resource utilisation 40
4.5 Button pad (5 buttons) resource utilisation 40

List of Figures

2.1 FPGA structure 9
2.2 Conceptual 4-input logic block 9
2.3 Generating bitstream 9

3.1 Connecting peripherals in MSB 22
3.2 Editing UART properties 23
3.3 LEON configuration 24
3.4 C/C++ wizard in Eclipse (LEON3) 25

4.1 Buttons on the development board 34
4.2 Signal with bounces 38
4.3 Debounce flowchart 39
4.4 Dual edge detector concept 39
4.5 Overview of the button pad device 41
4.6 Interrupt mechanism 41

A.1 Typical AMBA usage 60
A.2 WISHBONE interconnection types 61

Appendix A

Bus specifications

In this appendix a brief introduction is given to the two bus standards studied in this master's thesis.

A.1 AMBA Rev 2.0

The AMBA-2.0 specification defines an on-chip communications standard for designing high-performance embedded microcontrollers. The specification was released in May 1999 by ARM Limited[41]. Three buses are defined within the AMBA specification:

- AHB - Acts as the high performance system bus.
  - High performance
  - Pipelined operation
  - Multiple bus masters
  - Burst transfers
  - Split transactions
- ASB - Is an alternative to AHB, where AHB high performance features is not required.
  - High performance
  - Pipelined operation
  - Multiple bus masters
- APB - Bus intended for low-power peripherals.
  - Low power
  - Latched address and control
  - Simple interface

– Suitable for many peripherals

A typical solution based on AMBA is shown in Figure A.1

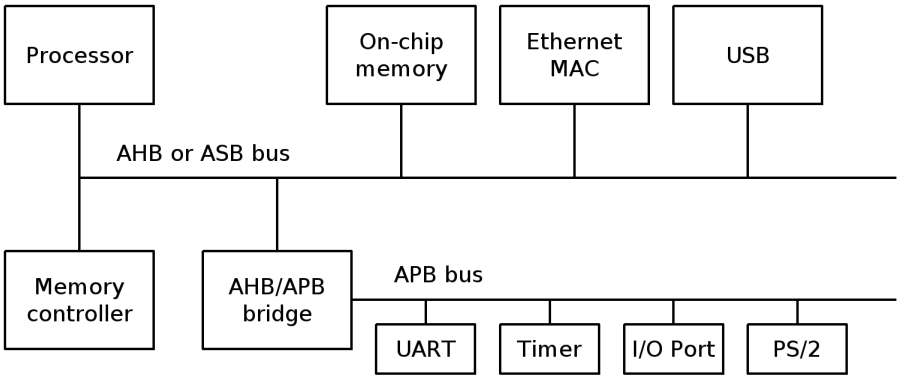

Figure A.1: Typical AMBA usage

For more information, see [40].

A.2 WISHBONE Rev B.3

WISHBONE is an open, freely useable SoC interconnection (master/slave) architecture for IP cores. The latest specification of the bus is revision B.3 and was released in September 2002. The intention is to use it as an internal bus for SoCs. WISHBONE is a general purpose interface so it only defines a standard set of signals and bus cycles to be used between IP core and does not attempt to regulate the application specific functions. Four types of interconnection (see Figure A.2) are defined:

- Point to point - A single master to a single slave.
- Data flow - Each IP core in the data flow has a master and slave interface.
- Shared bus - Connects two or more masters with one or more slaves.
- Crossbar switch - Connects two or more masters together so that each can access two or more slaves.

For more information, see [46].

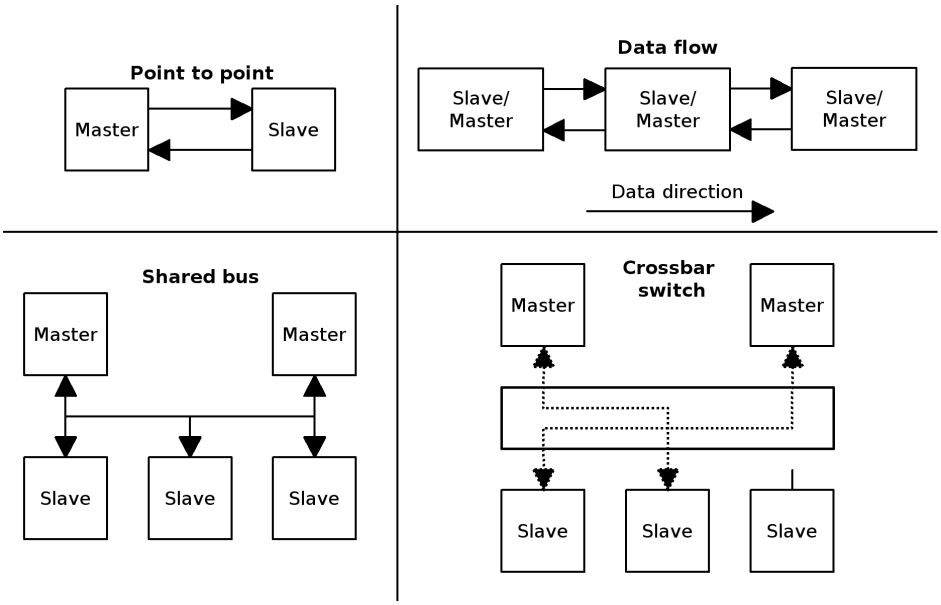

Figure A.2: WISHBONE interconnection types